A Survey of Practical Object Space Visibility Algorithms

Sherif Ghali
Max Planck Institute for Computer Science
Saarbrücken, Germany
ghali@mpi-sb.mpg.de

Contents

1 Introduction **1**

2 Motivation **2**

3 Taxonomy of Practical Object–Space Algorithms **3**

4 Edge–Polygon Clipping **3**

5 Quantitative Invisibility **4**

6 Polygon–Polygon Clipping **7**

7 Line Arrangement **7**

8 Segment Arrangement **9**

9 Trapezoidal Decomposition **10**

10 Conclusion **11**

1 Introduction

In a 1974 survey, Sutherland, Sproull, and Schumacker [23] presented the taxonomy of visibility algorithms shown in Fig. 1. They dubbed one branch of that taxonomy "object–space," to indicate that the algorithms in that category operate in object precision, as opposed to image precision. Four algorithms were described in that taxonomy and all were practical algorithms. Surprisingly, practical object–space visibility still offers challenges. We consider the known algorithms and describe these challenges.

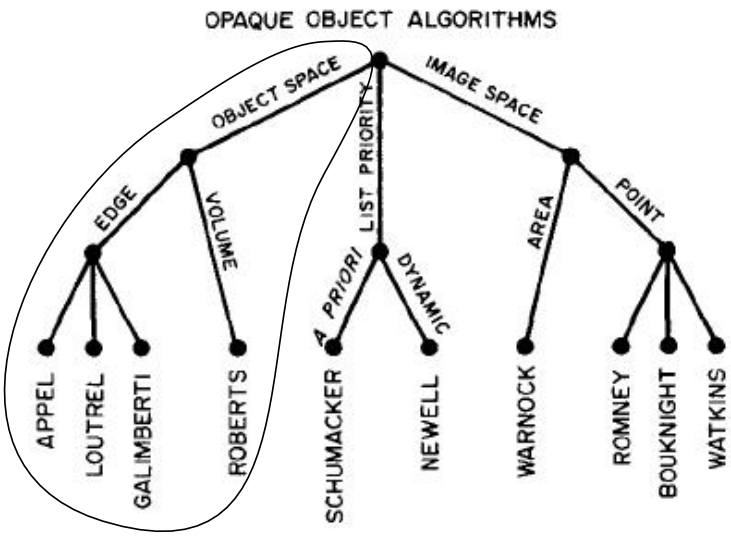

Figure 1: We consider the practical object–space visibility algorithms known to date. These algorithms fall in the left branch of Sutherland et al.'s classification [23].

Three versions of object–space visibility may be considered depending on the desired output. Given a collection of polygons, the location of a viewer, and a projection plane in space, the three versions report, in increasing order, the set of segments bounding the polygons that are visible by the viewer, the set of polygons visible, or the partition induced by the visible polygons on the projection plane.

The three problems are called, Hidden Line Elimination (or conversely, Visible Line Determination), Hidden Surface Elimination (or Visible Surface Determination), and Visibility Map Determination. Algorithms designed for one problem cannot always be recast to compute a solution for another. It is not necessarily possible to recast an algorithm for hidden line elimination, for example, to compute visible surfaces. We proceed by dividing the algorithms based on the approach they take and mention in the process which versions of the problems they address. The output of the three problems is illustrated in Fig. 2.

To avoid confusion, the usage of the word *practical* needs a clarification. An algorithm is practical if it appears that it is possible to implement it. In this sense this designation is somewhat subjective. It does not signify that a report on its implementation is available, but rather that the algorithm seems amenable to an implementation. There is in fact a dearth of reports on the implementation of object–space visibility algorithms. Aside from the four algorithms reported in Sutherland et al.'s survey, few other reports are known [24, 14, 3].

Non–practical, or theoretical, object–space visibility has a rich bibliography. We choose in this manuscript to omit discussing the theoretical algorithms and refer the interested reader to surveys

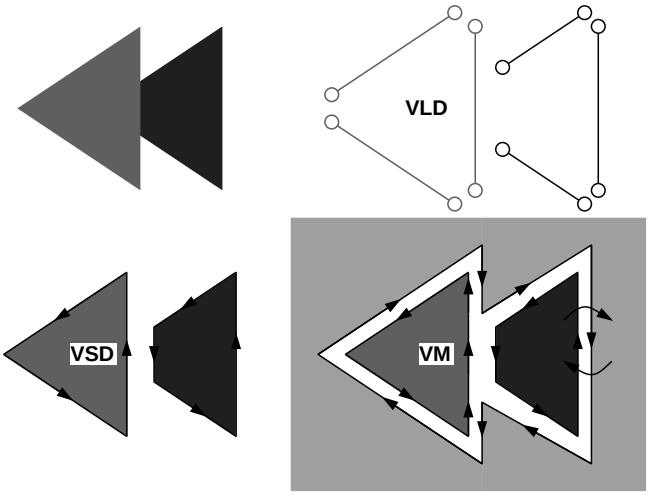

Figure 2: A view of two triangles; the visible lines; the visible surfaces; and the visibility map (only one edge adjacency is illustrated on the right).

that cover the theoretical algorithms. Dorward's and de Berg's expositions [9, 5] remain current. Durand's more recent survey [10] is broader in coverage, but is also somewhat succinct. Devai's survey [8] also covers aspects of visibility computation.

2 Motivation

Current computer graphics interactive techniques predominantly use raster methods for rendering, but there are several reasons why object–space visibility techniques remain relevant. One reason is that the output of an object–space visibility algorithm can be highly concise compared with its raster counterpart. An output possibly requiring hundreds of thousands of bytes using a raster description may be described in object space using a few hundred bytes. Output from object space algorithms can also be simply magnified at ease using standard viewers. All early developments in computer graphics were geared to vector displays and to plotters, both of which are vector output devices, or devices capable of drawing on a continuous rectangular area of the plane, rather than on a grid. The postscript language promotes the interest in vector–based or object space visibility algorithms.

Binary Space Partitioning, or BSP, trees are an object–space partitioning data structure. But no known technique uses BSP trees to solve any of the three object–space visibility problems described above. BSP trees can be used to produce an arbitrarily high resolution image, but such usage is limited to computing the depth order from the viewer, or a sorted order from far to near of (fragments of) the polygons in the scene. This list is then painted on a raster device to produce an image. The list before painting contains no information about what is visible and the amount of time needed to paint an image may be substantial if the depth complexity, or the number of overlapping polygons, is large. It seems quite plausible that BSP trees could be used to compute either hidden line removal or the visibility map, but no currently known algorithm does so.

Traditional methods for shadow computation rely on raster visibility methods. Such methods [25] have been criticized for computing jagged shadow edges and for requiring the user to fine tune the resolution of the shadow buffer with the resolution of the rendered image. Percentage–closer filtering [19] reduces the aliasing of the shadow edges at the cost of blurring them. Such

aliasing can be eliminated by using shadow volume BSP, or SVBSP, trees [4]. It is also possible to use another technique [11] that relies on true object–space visibility for the computation of alias–free shadows without the scene fragmentation obtained as a result of using BSP trees.

Recent non–photorealistic rendering techniques also need to compute object–space visibility. See, e.g., the work of Markosian et al. [15].

3 Taxonomy of Practical Object–Space Algorithms

The known practical algorithms can be divided into the following categories:

Edge–Polygon Clipping Clip each edge against each polygon.

Quantitative Invisibility Incrementally update quantitative invisibility.

Polygon–Polygon Clipping Clip each polygon against each polygon.

Line Arrangement Extend edges to lines and compute the line arrangement.

Segment Arrangement Compute the view plane partition (also called the subdivision graph) and traverse the resulting regions by crossing over the edges.

Trapezoidal Decomposition Compute the view plane partition by incrementally adding the polygons.

For simplicity and unless otherwise noted in this discussion, we assume that the view transformation (including perspective) has been applied, that the input has been clipped against the (normalized) view volume and that the projection is orthogonal to the xy–plane with the viewer located on the positive z–axis looking towards the negative z–axis. We also assume that a right–hand coordinate system is used for all computation. Finally, the list of edges defining a polygon are listed in counter–clockwise order around each polygon such that the polygon is to the left of an edge when moving along the directed edges defining the boundary. The conventions adopted sometimes deviate from those above in two respects and so we justify the choice taken here. If the viewer is located at $(0, 0, -\infty)$ and is looking towards the positive z–axis, a transformation to left–handed coordinate system is necessary, since otherwise the origin of the projection would lie on the lower right–hand corner, which would complicate the vector manipulations. Also, the edges defining a polygon are sometimes assumed to run clockwise around a polygon. The counter–clockwise order is preferred since it is then possible to calculate the cross–product of two consecutive edge vectors in a convex polygon to determine the polygon normal. If the polygons define the boundary of a closed solid, the polygon normals are directed towards the outside of the solid. A standard computer graphics books (such as Hearn and Baker's [13]) provides a good introduction to these topics.

4 Edge–Polygon Clipping

The earliest known method for hidden line elimination is due to Roberts [20]. That technique considers every edge in the scene and compares it to every polygon. The portions of the edge that remain after purging those that are hidden by one or more polygon are output as the visible edges. This leads to the following brief pseudo–code.

for each edge e
    for each polygon p
        eliminate portion of e hidden by p

Rather than iterate over individual polygons, Roberts's algorithm iterates over the solids and in so doing restricts the input to convex solids. The intersection operation between an edge and a polygon is carried out by parameterizing the set of line segments from the viewer to the edge in question. If this line of sight is obstructed by the solid, a subset of the edge is found to be hidden from the viewer and needs to be eliminated. See Figure 3. Roberts defines the set of points inside a solid using a collection of inequalities based on the equations of the planes in which the solid's polygons lie. This leads to the following.

E $\leftarrow$ set of edges belonging to front–facing polygons
while E is not empty **do**
    e $\leftarrow$ E.getElement()
    for each solid s
        E $\leftarrow$ E $\cup$ fragment(s) of e that remain visible
output edges in E as visible from the viewer

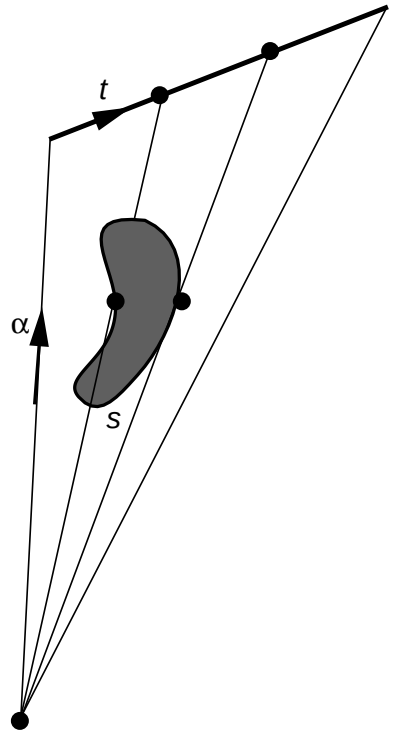

Figure 3: The lines of sight from the viewer to the edge are parameterized by t and α. The subset of the edge for which these lines of sight intersect the solid s are hidden from the viewer.

5 Quantitative Invisibility

The notion of quantitative invisibility, due to Appel [1], appears repeatedly in visibility algorithms under different names. For a point P on an edge in the scene and a viewer V, the quantitative

invisibility of P is the number of intersections of the line segment PV with the polygons in the scene. The point P is visible from V if and only if its quantitative invisibility (QI, for short) is zero. The idea is that it is sufficient to compute QI for an edge at one endpoint and update its value along the edge. To do so one intersects the edge with only the contour edges of the solids in the scene since it is at those contour edges that the QI can change. Also, since the QI at a vertex is the same for all adjacent edges (except for a detectable special case), it is sufficient to compute the QI one vertex per solid and propagate the value around the edges of the solid. See Figure 4.

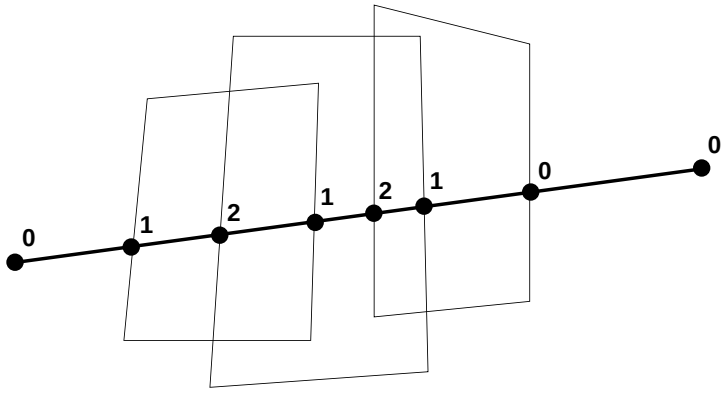

Figure 4: The quantitative invisibility of one vertex on each solid is determined. This QI is incrementally updated along each edge on the solid and the regions of the edge with QI equal to zero are reported as visible. In the figure, only the two extreme portions of the edge e are output.

R $\leftarrow$set of scene edges */* relevant edges */*
C $\leftarrow$set of contour edges
for each solid s
  pick an initial vertex P
  QI $\leftarrow$quantitative invisibility of P
  / Traverse the edges on s */*
  for each edge e in s starting from an edge adjacent to P
    intersect edges in C with e
    sort the resulting intersections along e
    evaluate QI incrementally along e
    output the fragments whose QI is zero

Appel's technique does not tackle the boundary cases, such as the case of a vertex lying on the projection of the edge under consideration illustrated in Figure 5. Blinn [3] defines the notion of *fractional invisibility*. In the two cases shown in Figure 5 and assuming that such cases can be reliably detected, he increments or decrements the quantitative invisibility by $\frac{1}{2}$. Thus in the figure on the left the two halves will add to 1 and the right hand part of the edge will be correctly identified as invisible whereas in the figure on the right one increment and one decrement operations will leave the value of QI unchanged.

Another method relying on QI is due to Goodrich [12] who describes two visibility algorithms in that work, one for hidden lines and the other for hidden surface computation. The two algorithms use different solution techniques so they are listed here in two categories.

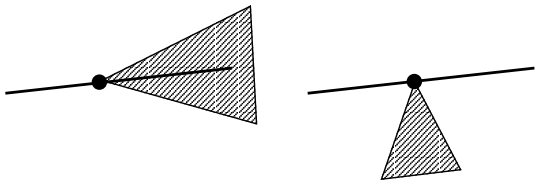

Figure 5: Rather than increment or decrement the quantitative invisibility by 1, Blinn [3] uses a fraction to handle boundary cases.

Goodrich also computes and propagates quantitative invisibility (which he calls *coverage*), but he makes use of the following two observations:

1. It is sufficient to initialize the QI for a small number of vertices, called *representative vertices*. In the arrangement of the projection of the scene polygons, each connected component has only one representative vertex.

2. To initialize the QI at a representative vertex, it is not necessary to shoot a ray from the vertex to the viewer and compute an expensive segment–polygon intersection for the set of input polygons, instead it suffices to traverse the arrangement just described and to maintain a list of the polygons in whose interior the traversal is. This small set of polygons (which is related to the depth complexity of the scene) is the one queried when computing the QI of a representative vertex.

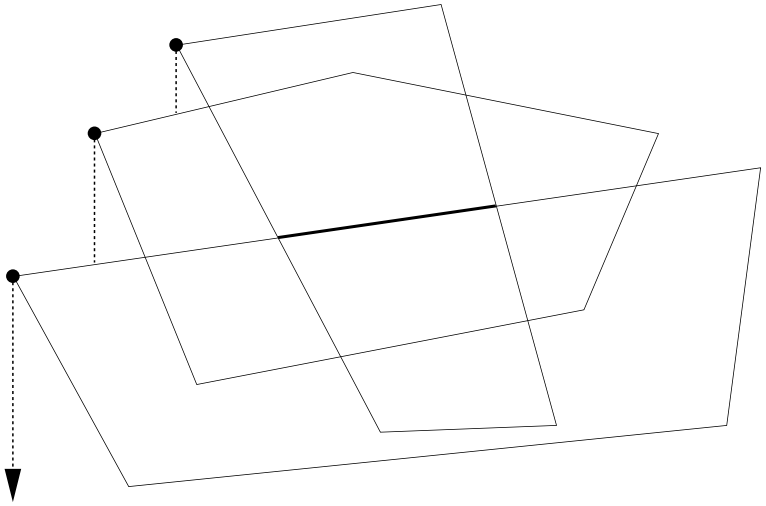

Figure 6: The three polygons shown in the figure are projected on the view plane and their arrangement is computed. The representative vertices are highlighted. Each representative vertex is connected to another polygon (if any) in a selected direction (here, negative–y). During the traversal of the highlighted edge for example, we shall know that we are inside all three polygons (in the view plane projection).

Goodrich's algorithm proceeds in the following three steps:

1. Project the front–facing polygons on the view plane and compute their arrangement. Minimize the number of connected components in that arrangement by shooting a ray from the leftmost vertex of each solid in one direction (e.g. negative–y) on the projection plane

2. Traverse the arrangement while maintaining the list of polygons that we enter or exit. When a representative vertex is reached, this list of polygons is queried to determine the QI of the vertex.

3. Incrementally update the QI starting from each representative vertex by traversing the arrangement.

We do not list pseudo–code here, but rather refer the interested reader to the one given by Hostetler [14].

6 Polygon–Polygon Clipping

The first algorithm to compute true hidden surface elimination was described in 1977 [24]. Prior algorithms had used a combination of depth sorting and painting, but as with BSP painting mentioned earlier, the output from such algorithms is a raster output since the vital visibility determination operation is performed at pixel precision. Weiler–Atherton's algorithm starts by computing an initial depth sort of the polygons by proximity from the viewer. An iteration is then started in which the nearest polygon is clipped against the rest of the polygons in the list. The polygons that lie inside the clip polygon are discarded (or are taken as polygons lying in shadow in the sequel paper [2] if visibility is computed from the location of a light source) and the polygons lying outside the clip polygons are processed further.

Determining an initial depth sort is itself not a trivial operation not the least since a depth sort for the set of input polygons may not exist. The sort is performed using an arbitrary key, such as the vertex on each polygon nearest to the viewer. After each clipping operation, this initial sorting order is checked for correctness. If a polygon in the inside–clip list is found to be closer to the viewer than the clip polygon, then an error is detected in the initial sorting order. The authors report that they use recursion to handle that case.

```
L ←determine depth order of input polygons (nearest to farthest)
    /* use the nearest vertex of each polygon as the sorting key */
while L has more than one element do
    p1 ←L.extractFirstPolygon()
    /* clip p1 with the polygons in L */
    insideList ←portions of polygons in L inside p1
    outsideList ←portions of polygons in L outside p1
    /* discard insideList */
    L ←outsideList in the same initial sorting order
report polygons remaining in L as visible
```

7 Line Arrangement

Two algorithms fall in this category and both represent milestones in visibility algorithms as both break a theoretical barrier from $O(n^2 \log n)$ to $O(n^2)$ to process a scene of n edges. Both the algorithm of Devai [6] and that of McKenna [16] are practical in the sense that they are implementable (Hostetler [14] reports on his implementation of Devai's algorithm). Unfortunately, neither algorithm is likely to be useful in practice. The reason for this is that the first step taken by both algorithms is to extend the edges defining each input polygon to straight line and to process the intersections

of these lines on the projection plane. Thus even for a scene where all edges are relatively small (for example a large number of toothpicks that fell on a floor, orthogonally viewed on that floor) extending the edges to lines means that the algorithm is guaranteed to spend time quadratic in the number of edges (even if each toothpick intersects only few others).

Devai observes that the initial computation of quantitative invisibility can be avoided after the edges are extended to lines. In that case, it is sufficient to start with a value of QI at zero (at infinity) and to process the intersections as before. Only segments deemed visible and that overlap with the original input segment are reported as visible. See Figure 7.

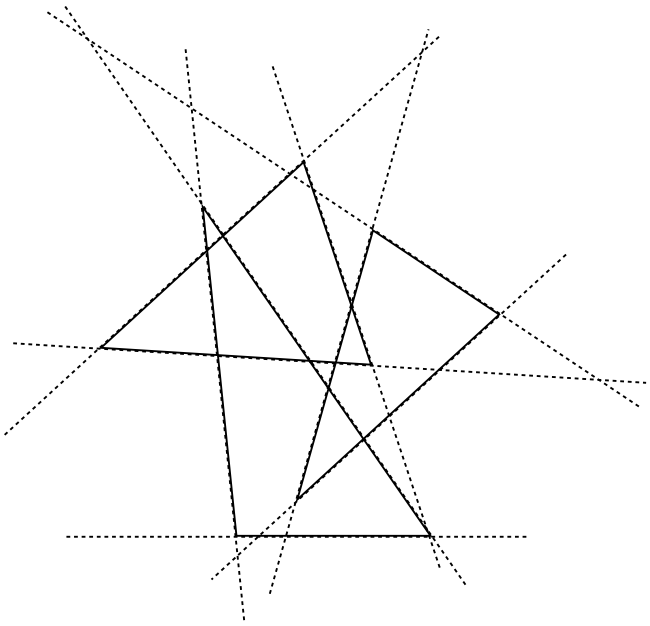

Figure 7: In Devai's algorithm, to determine the visible segments in this scene consisting of three triangles, each line segment defining the boundary of a triangle is extended to a line. The intersections of these lines on the projection plane are handled using the QI increments/decrements.

The pseudo–code of Devai's algorithm follows.

S $\leftarrow$set of lines resulting from extending the scene edges
for each line g in S
    intersect the lines in S with g
    initialize the quantitative invisibility of g to zero (at infinity)
    propagate QI along the intersections
    output fragments inside the original edge and with zero QI

McKenna also extends the line segments to straight lines, but rather than handle the intersections along each line independently, the intersections are handled by sweeping a (vertical bendable) line from left to right. Also, rather than processing the intersection points in left–to–right order, the notion of *completable regions* is introduced. This is illustrated in Figure 8.

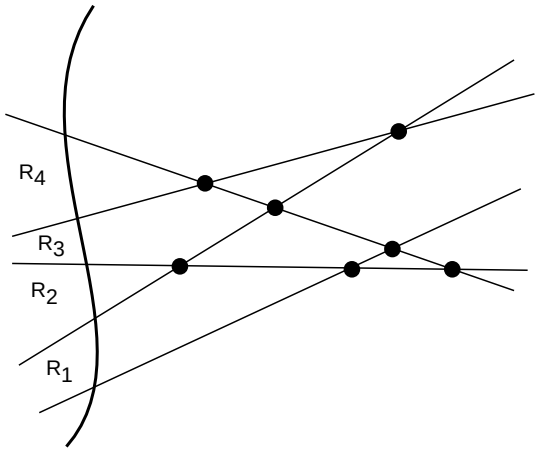

Figure 8: Following McKenna, the sweep line (called bendable line or topological line) is illustrated in bold and the regions thus far completed are labeled R_1 to R_4. Regions R_2 and R_4 are called completable regions because its two (upper and lower) bounds intersect the topological line and intersect each other at a vertex. The collection of such vertices are stored in an unordered queue.

8 Segment Arrangement

Algorithms that extend the line segments on the projection plane to complete lines and then compute their intersections are said not to be *intersection sensitive*. This means that even though a pathological scene generating the square of the input number of polygons on the view plane can be described and would thus force any algorithm to take time quadratic in the input, it is desirable to have an algorithm that is intersection sensitive in the sense that if fewer than quadratic intersections in the number of input edges appear on the view plane, the algorithm would spend time less than quadratic as well. Goodrich describes such an algorithm [12]. The time complexity of his algorithm is proportional both to the number of intersections on the view plane as well as to the depth complexity of the scene, or the maximum number of polygons intersected by a ray issued from the location of the viewer. He computes the arrangement of the line segments as projected on the view plane and uses the result to compute the visible *surfaces*. This method is described below.

1. Project the polygons and compute the arrangement of the projected edges.
2. Determine the depth order of the polygons. Two steps are needed to do so.
   (a) Each intersection of two edges in the arrangement provides information about the relative order of the two polygons to which the edges belong. Construct a directed graph in which each polygon is a node and there is an edge from each polygon to those it occludes given that occlusion relationship.
   (b) The depth order is given by a topological sort of the resulting directed graph. If a cyclic order is detected, the algorithm halts and reports the offending sequence for the user to break some polygons in the sequence before restarting.
3. From back to front order, and simulating the painter's algorithm in object–space, traverse the boundary of each polygon. During this traversal, the edges that lie inside the polygon and that belong to polygons farther from the viewer are marked as invisible. See Figure 9.

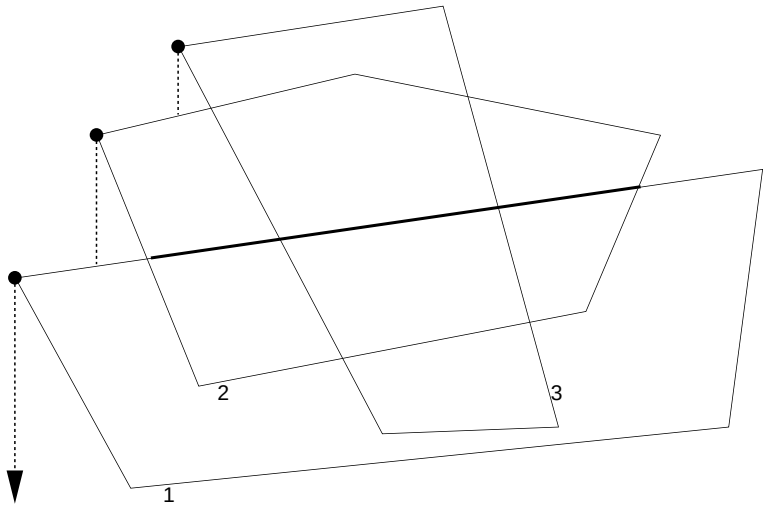

Figure 9: In this scene consisting of three polygons labeled in increasing order from farthest to nearest and after polygon 1 has been processed, during the processing of polygon 2, the highlighted is marked as invisible as it is overwritten by polygon 2 in this simulation of a painter's algorithm in object–space.

Another technique that also relies on computing the partition of the plane induced by the projection of the scene edges is a visible surface determination algorithm (which can also be used to compute the visibility map) described by Devai [7]. After computing the segment arrangement, he proceeds by traversing the regions of that partition. Traversal of the regions is performed by crossing over each *edge* in the partition and each edge thus crossed either inserts or deletes one polygon from a set of polygons that are maintained and that consist of the polygons in whose interiors the region lies. These polygons are maintained in a priority queue and the top face of the queue is the polygon visible at the currently visited region.

Schmitt's [21, 22] describes techniques that also rely on the computation of segment arrangement.

9 Trapezoidal Decomposition

Algorithms that start by computing the arrangement of line segments on the view plane and then purge the hidden edges perform poorly on a large class of inputs that arise frequently in practice. If a scene database consists of a building and one renders an image from a room in that building such that the complexity of the visible image is relatively small, then an algorithm that starts by calculating the set of edge intersections on the image plane is wasteful as it would be more desirable to have an algorithm that runs in time proportional to the size of the output, or an *output–sensitive* algorithm for short. No practical algorithm is known that approaches that objective. Mulmuley describes a promising, though unimplemented, approach [17] (also described elsewhere [18]). The crux of his idea is that it is a waste to compute the intersection of two edges if the resulting intersection is to be hidden by polygons closer to the viewer. Rather than compute the complete set of intersections, he incrementally computes the view by randomly adding the polygon in the scene to construct a new view. A problem with such an incremental approach is that there is no way to know with which set of edges the newly inserted polygon should be intersected and iterating in search of those intersections would defeat the savings intended. Instead, use is made of a set of

conflicts. These conflicts are a relation between regions on the image plane and not–yet handled polygonal vertices. When inserting a polygon in the view, conflicts give a cheap method to identify the region in which to insert an endpoint.

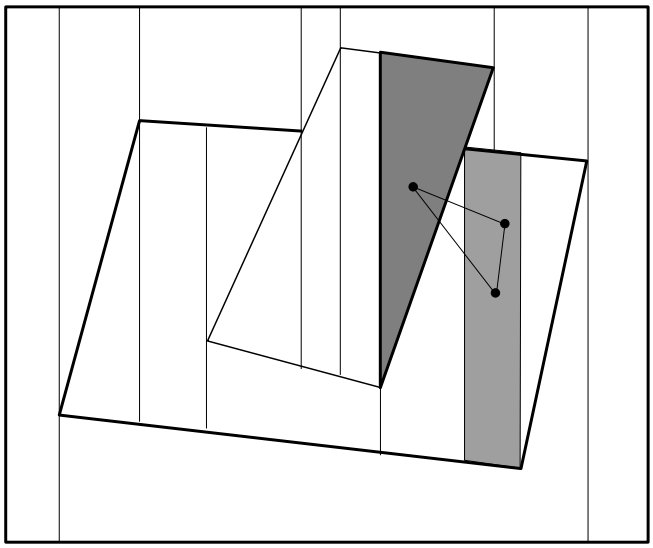

Figure 10: A partition of the view plane is incrementally constructed. The figure shows the stage when two polygons are already in this trapezoidal decomposition and a third polygon, a triangle, is to be inserted. To efficiently determine the regions in the partition where the triangle's vertices should be inserted, a conflict is maintained between the regions and the vertices of the not–yet–processed polygons.

10 Conclusion

It is clear that the main reason why object–space techniques are no longer pursued as a topic of research is that interactive techniques heavily rely on raster methods. As outlined earlier, there are, however, many reasons why object–space methods remain relevant. Perhaps the most important objective is simply as an intellectual challenge in the pursuit of a classical problem in computer graphics that remains open (a practical output–sensitive algorithm). Also, raster rendering falls short of being satisfactory in many common applications. The resolution of current print devices, for example, far exceeds the resolution of the images rendered in technical documents. It is possibly a holy grail to revise the raster rendering techniques, which have been devised for screen viewing, with object–space rendering techniques suitable for inclusion in printed documents. Even if in preparing a printed document one fully uses the available print device resolution, a document and the printed representation of a document should not be tied together. Namely, a rendering of a document produced today on printing devices available several years from now would fully use the resolution of these devices if the images in the document are prepared using object–space methods.

Bibliography

[1] A. Appel. The notion of quantitative invisibility and the machine rendering of solids. In *Proc. ACM National Conf.*, pages 387–393, 1967. also published in Rosalee Wolfe, ed., Seminal Graphics, Pioneering Efforts that shaped the Field, ACM Press, 1998, pp. 19–26.

[2] P. Atherton, K. Weiler, and D. P. Greenberg. Polygon shadow generation. *Computer Graphics*, 12(3):275–281, 1978. Proc. SIGGRAPH '78.

[3] J. Blinn. Fractional invisibility. *IEEE CG&A*, November 1988. also available in *Jim Blinn's Corner, A Trip Down the Graphics Pipeline*, Chap. 10, Morgan Kaufman.

[4] N. Chin and S. Feiner. Near real–time shadow generation using BSP trees. In *SIGGRAPH '89*, pages 99–106, August 1989.

[5] M. de Berg. *Ray Shooting, Depth Orders and Hidden Surface Removal*, volume 703 of *Lecture Notes Comput. Sci.* Springer-Verlag, Berlin, Germany, 1993.

[6] F. Dévai. Quadratic bounds for hidden line elimination. In *Proc. 2nd Annu. ACM Sympos. Comput. Geom.*, pages 269–275, 1986.

[7] F. Dévai. An intersection-sensitive hidden-surface algorithm. In G. Marechal, editor, *Eurographics '87*, pages 495–502. North-Holland, August 1987.

[8] F. Dévai. On the computational requirements of virtual reality systems. In *Eurographics 1997 State of The Art Report*. Eurographics, 1997.

[9] S. E. Dorward. A survey of object-space hidden surface removal. *Internat. J. Comput. Geom. Appl.*, 4:325–362, 1994.

[10] F. Durand. *3D Visibility: Analytical Study and Applications*. PhD thesis, Université Joseph Fourier, Grenoble, France, 1999.

[11] S. Ghali. *A Geometric Framework for Computer Graphics Addressing Modeling, Visibility, and Shadows*. PhD thesis, Department of Computer Science, University of Toronto, 1999.

[12] M. T. Goodrich. A polygonal approach to hidden-line and hidden-surface elimination. *CVGIP: Graph. Models Image Process.*, 54(1):1–12, 1992.

[13] D. Hearn and M. P. Baker. *Computer graphics*. Prentice Hall, 2nd ed. edition, 1994.

[14] L. B. Hostetler. A comparison of three hidden line removal algorithms. Report TR–89–02, Department of Computer Science, Johns Hopkins University, 1989.

[15] L. Markosian, M. A. Kowalski, S. J. Trychin, L. D. Bourdev, D. Goldstein, and J. F. Hughes. Real-time nonphotorealistic rendering. *Computer Graphics*, 31(Annual Conference Series):415–420, August 1997.

[16] M. McKenna. Worst-case optimal hidden-surface removal. *ACM Trans. Graph.*, 6:19–28, 1987.

[17] K. Mulmuley. An efficient algorithm for hidden surface removal. *Computer Graphics*, 23(3):379–388, 1989.

[18] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice Hall, Englewood Cliffs, NJ, 1994.

[19] W. T. Reeves, D. H. Salesin, and R. L. Cook. Rendering antialiased shadows with depth maps. *Computer Graphics*, 21(4):283–291, July 1987.

[20] L. Roberts. Machine perception of three-dimensional solids. In J. Tippett et al., editors, *Optical and Electro-Optical Information Processing*, pages 159–197. MIT Press, Cambridge, MA, 1965. also published as MIT Lincoln Laboratory, TR 315, (May 1963).

[21] A. Schmitt. On the time and space complexity of certain exact hidden line algorithms. Report 24/81, Fakultät Inform., Univ. Karlsruhe, Karlsruhe, West Germany, 1981.

[22] A. Schmitt. Time and space bounds for hidden line and hidden surface algorithms. In *Proc. Eurographics 81*, pages 43–56, Amsterdam, Netherlands, 1981. North-Holland.

[23] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker. A characterization of ten hidden-surface algorithms. *ACM Comput. Surv.*, 6(1):1–55, March 1974.

[24] K. Weiler and P. Atherton. Hidden surface removal using polygon area sorting. *Computer Graphics*, 11(2):214–222, 1977. Proc. SIGGRAPH '77.

[25] L. Williams. Casting curved shadows on curved surfaces. *Computer Graphics (SIGGRAPH '78 Proceedings)*, 12(3):270–274, August 1978.